Modern Engineering

January 23 | Day 10

Prudential Financial

GENERAL ASSEMBLY

Whilst we wait....

In chat write:

Your name - role and a recent impulse buy.

Let’s give everyone a min or two to join

LESSON ROADMAP

- Test Driven Development
- Unit Testing in JavaScript
- Create Unit Tests Lab
- API Testing with SuperTest
- Create API Tests

MEF MODULE 4 DAY 10: Test Driven Development and API Testing

Schedule	
9:00–9:15 am	Welcome and Warm-Up
9:15–10:00 am	Test Driven Development
10:00 am–10:30 pm	Introduction to Jest
10:30–12:30 pm	**Writing Unit Tests with Jest**
12:30–1:30 pm	Lunch
1:30-1:30 pm	Preparing Express to be Tested
1:30–4:50 pm	**Replacing Postman with SuperTest**
4:50–5:00 pm	Bring It Home

LEARNING OBJECTIVES

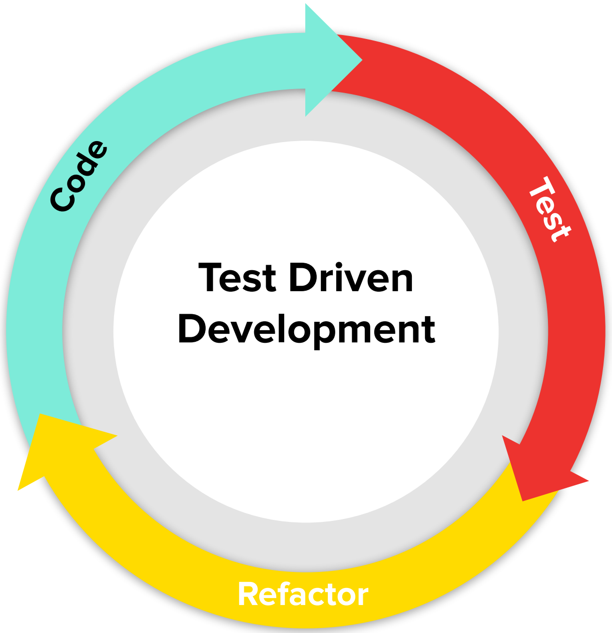
1. Use effective git branching strategies
2. Articulate the benefits of Test Driven Development and compare unit, integration, and end to end tests
3. Design unit tests using the Jest framework
4. Design tests for an API using SuperTest

Modern Engineering

Test Driven Development

What is Test Driven Development (TDD)?

Test-Driven Development is an **iterative programming process consisting of 5 steps** that involve writing a series of unit tests to capture the requirements of a feature and then writing the code for those tests to pass.



Test-Driven Development is *not* a software development methodology but a testing methodology and can be used in conjunction with any development methodology such as Agile/Scrum and waterfall methodologies.

TDD: Pros and Cons

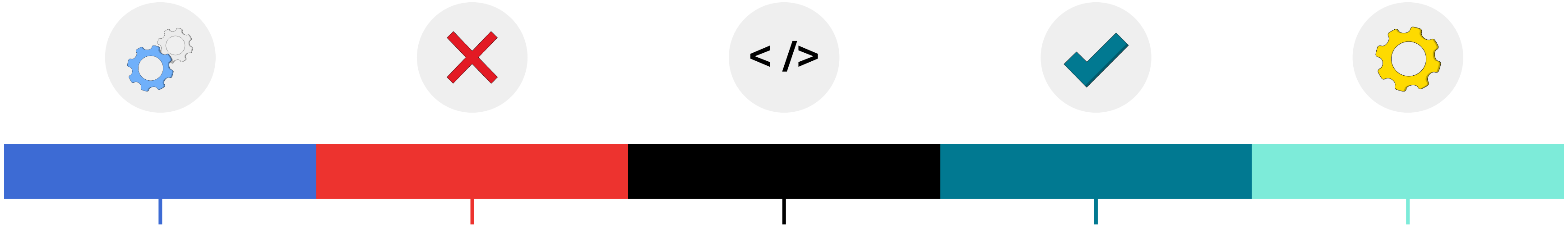
Pros

- Only write necessary code
- Focus on modular design
- Refactor friendly
- A complete declarative test suite
- Less debugging

Cons

- All or nothing
- Not a silver-bullet
- Requires extra time

5 steps of the TDD life-cycle



Step 1

Create tests

Step 2

Run tests
(all tests should **fail**)

Step 3

Write feature
code

Step 4

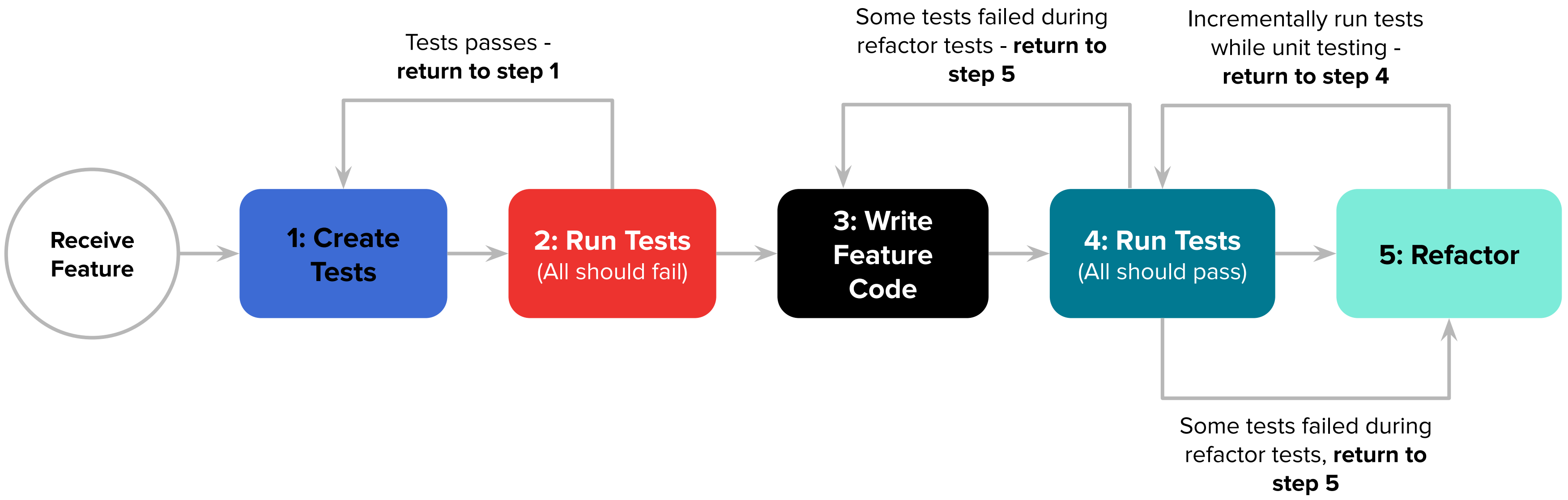
Run tests
(all tests should **pass**)

Step 5

Refactor

TDD Testing Life-Cycle

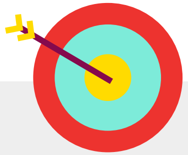
The goal is to develop our software from a tests-first perspective, using our test cases as a guide to ensure feature requirements and having a set of guard rails to keep our code as simple as possible while maintaining a fast-pace of development. **Let's walk through the steps together.**


Tests passes - **return to step 1**
Some tests failed during refactor tests - **return to step 5**
Incrementally run tests while unit testing - **return to step 4**
Receive Feature
1: Create Tests
2: Run Tests (All should fail)
3: Write Feature Code
4: Run Tests (All should pass)
5: Refactor
Some tests failed during refactor tests, **return to step 5**

TDD Is Not All About Test Coverage

It is a **common misconception** that the goal of TDD is to ensure total test coverage of one's projects.

- This is a benefit of TDD, but not inherently the primary goal of TDD
- One might be hard-pressed to determine a truly valuable justification for 100% test coverage

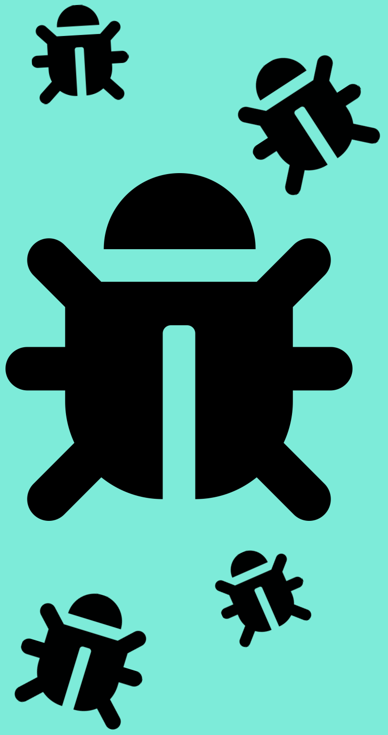


The goal of TDD is to foster the development of clean, DRY, domain-specific code bases that adhere to best practices most notably the singular-responsibility principle.

Automated Testing

Any sufficiently complex program is virtually guaranteed to have bugs. As software developers, a large portion of our time will be devoted to identifying, fixing, and preventing these bugs.
What if we could write programs that automated this verification for us?

By investing a small amount of time now to write these test programs, we could save time (and money) by *not* repeating these verifications manually.



Testing focuses for microservices

Unit Testing
Most relevant for TDD today

Component Testing

Integration Testing

Functional Testing

Unit Testing

Unit testing is the practice of testing individual pieces or units of source code to ensure that they behave as intended under various scenarios.

A unit should only be responsible for a singular concern.

EXAMPLE:
function sumTwoIntegers(integer1, integer2)

Test 1: sumTwoIntegers should create an error when we only enter 1 integer

Test 2: sumTwoIntegers should return 7 when the inputs 3 and 4 are passed to it

Test 3: sumTwoIntegers should create an error when we pass a string as one of its 2 parameters

Unit testing: Pros and Cons

Pros

- Encourages modular programming
- Can prevent many bugs if done well
- Increases developer productivity

Cons

- Can be time consuming
- Isn’t going to catch all bug
- Can be done badly and serve little value

Integration Testing

Integration testing is similar in theory to unit testing; however, the goal of integration testing is to ensure that `units` of code work together as expected. **Let's look at an example of what an integration failure could look like.**

Example Context

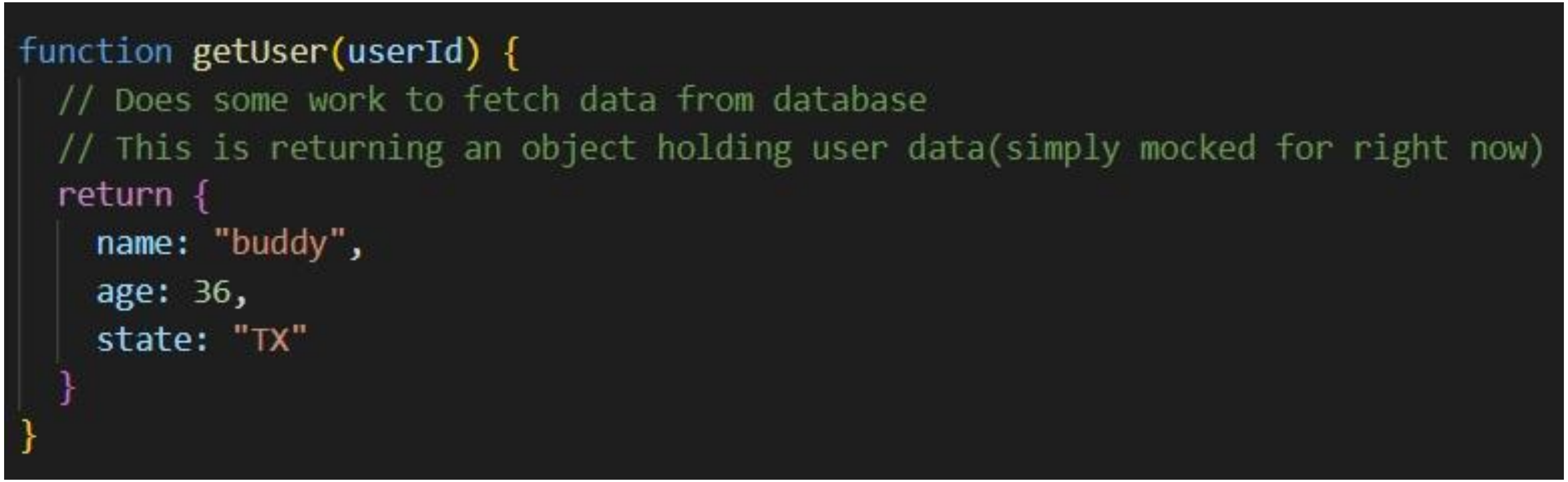
Data layer services integrate into a higher-order service, like a user service.

Data layer service has a method called 'getUser' that takes a single string parameter for userId.

Our userService has our data layer service as a dependency and uses the **'getUser'** function to produce user data.

Integration Testing Example pt. 1

Data-layer service **getUser** state initially:

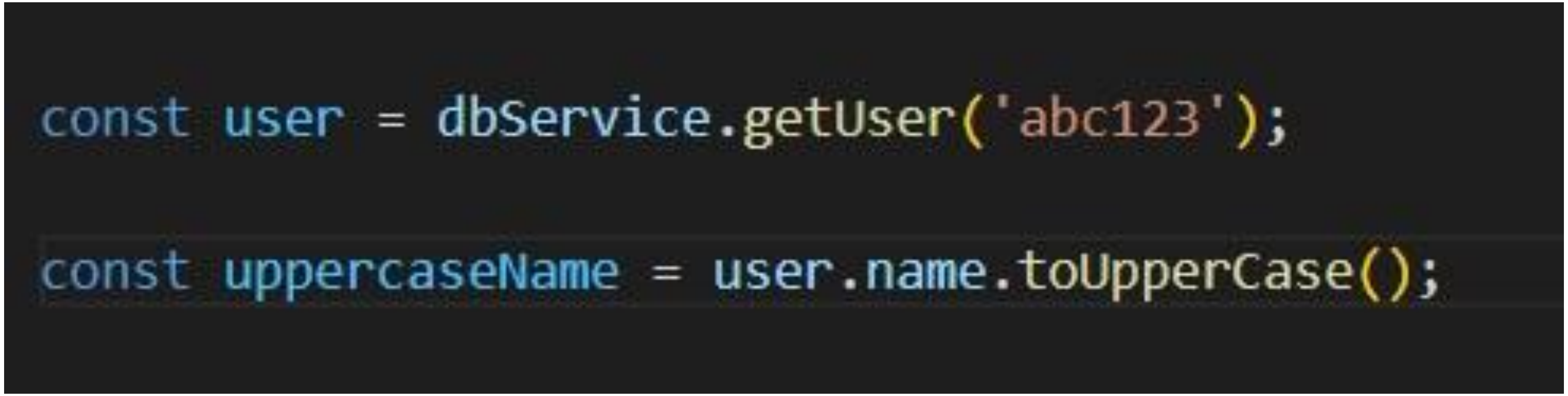


```javascript
function getUser(userId) {
  // Does some work to fetch data from database
  // This is returning an object holding user data(simply mocked for right now)
  return {
    name: "buddy",
    age: 36,
    state: "TX"
  }
}
```

Integration Testing Example pt. 2

Notice how the userService is expecting the data-layer service to return an object with a property of ‘name’ that should be a string.

Example of how userService may use ‘getUser’



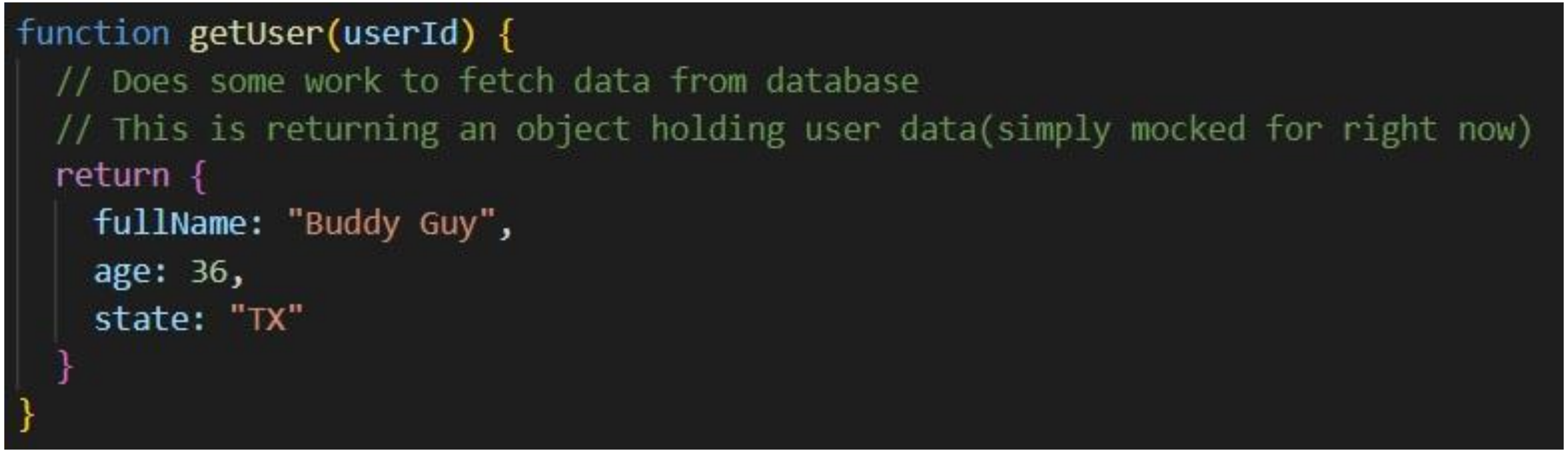
```
const user = dbService.getUser('abc123');

const uppercaseName = user.name.toUpperCase();
```

Integration Testing pt. 3

💡 What would happen if one of our team members modifies data service to change its return data, but neglects to consider areas of the project using 'getUser'?

Example of a breaking edit:



```
function getUser(userId) {
  // Does some work to fetch data from database
  // This is returning an object holding user data(simply mocked for right now)
  return {
    fullName: "Buddy Guy",
    age: 36,
    state: "TX"
  }
}
```

Integration Testing pt. 4

You’ll notice that our userService code expects for the ‘getUser’ method to return a property of ‘name’ will not run into runtime exception because that response no longer contains a property of ‘name’, it was replaced with ‘fullName’.
This is an simple example of what types of issues an integration test is intended to account for, <u>the interfaces between components.</u>

Integration testing: Pros and Cons

Pros

- Can verify if modules perform as expected in unity
- Verify that changes in modular contracts can break other areas of an application

Cons

- Testing too many components in the same test can cause fragile and unhelpful tests *BIG Bang*
- Only catches integration bugs

Component Testing

Component tests capture expected behavior of units of work in conjunction. The **goal of component testing** is to run automated tests that perform a full feature set of behaviors and verify that all singular modules interplay in the intended manner.

Component Testing: Pros and Cons

Pros

- Allows for workflow validation between sets of modules
- Catches a lot of common bugs
- Provides greater confidence in code base

Cons

- Can limit development freedom
- If done poorly can cause dependency hell in your test suite

End-To-End Testing

- **End-to-End testing** is the process of testing whole pieces of functionality in a product from start to finish.
- Ephemeral Environments (a short-lived, encapsulated deployment of an application.)
- End-to-End Testing efforts usually involve using additional testing frameworks like cucumber, selenium, or puppeteer and a headless browser.
- End-To-End tests are not often maintained by the same engineers that write the feature code and instead are maintained by QA engineers.

End-To-End Testing: Pros and Cons

Pros

- Automated full testing of key product features
- Can catch ample debugs that would be directly experienced by end users

Cons

- Expensive
- Fragile
- May require a dedicated resource to maintain

Recap

- Asynchronous concerns
- Environment setup

?

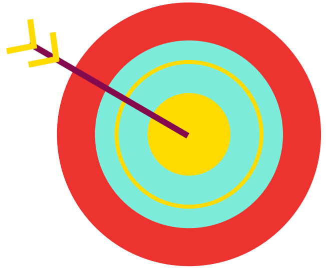
Any Questions

Modern Engineering

Introduction to Jest

What is Jest?

Jest is an easy-to-configure testing framework built by Facebook for testing JavaScript code. When in watch mode, Jest runs tests automatically.



The goal of these tests are to:

- **Establish** an expected result
- **Invoke** the unit of work we want to put under test to get an actual result
- **Compare** if the actual results meets our expected results

Guided Walk-Through:

What is Jest?

Let's say you have a .js file with several functions inside it that you wish to test. The easiest way to make them accessible to other files is to export the pieces of functionality you wish to be tested.

As a class, we'll walkthrough **forking** and **cloning** in a Jest TDD-practice-exercise.

Source: Jest

Setting-Up Jest Part 1

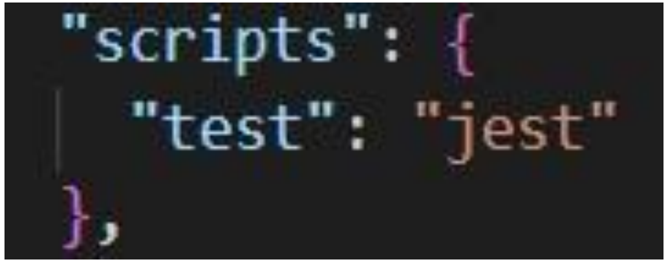
A basic set-up for Jest goes as follows:

1. Run **npm i jest –save-dev** in your project folder.
2. Create a **test** folder at the root of your project folder.
3. Jest will look for files that have **.spec** in them inside a folder named **test** or **tests**.

Setting-Up Jest pt. 2

4. Add your test script command to your package.json file like:

This will allow tests to be run using the **npm test** script command.

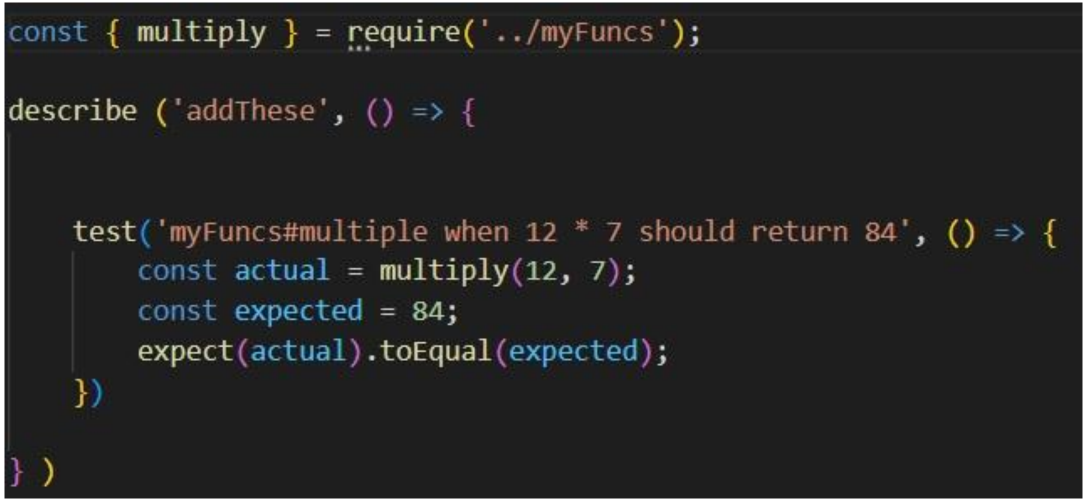


```
"scripts": {
    "test": "jest"
},
```

Image 12: Setting-Up Jest Part 2

Anatomy of Jest Testing

Below is the simplest version of a Jest unit test from our demo project. We will walk through the key parts of this and expand as we go.



```
const { multiply } = require('../myFuncs');

describe ('addThese', () => {


    test('myFuncs#multiple when 12 * 7 should return 84', () => {
        const actual = multiply(12, 7);
        const expected = 84;
        expect(actual).toEqual(expected);
    })

} )
```

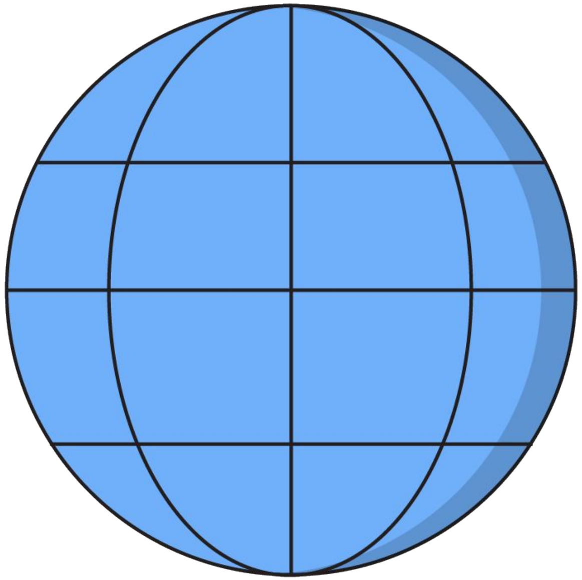
Image 13: Anatomy of Jest Testing

Take note:

- We are importing our functional code at the top
- We want to test our unit of work with the function multiply.
- Describe function
- Test function
- Expect function

Jest Components: Globals

- **Jest Globals** are core functions of test that are globally available when running Jest code.
- You do not need to import them into your test files.
- Some commonly used Globals:
    - Describe
    - Test
    - BeforeEach
    - afterEach



Source: Jest Globals

Jest components: Expect

- **Jest Expect** is what we will use to determine if our test pass or fail.
- Expect has two primary types of utility functions it uses to verify complex assertions:
    - Modifiers: .not, .resolve, .reject
    - Matcher: .toBe, toBeDefined, .toBeFalse, .toEqual, and among many others.

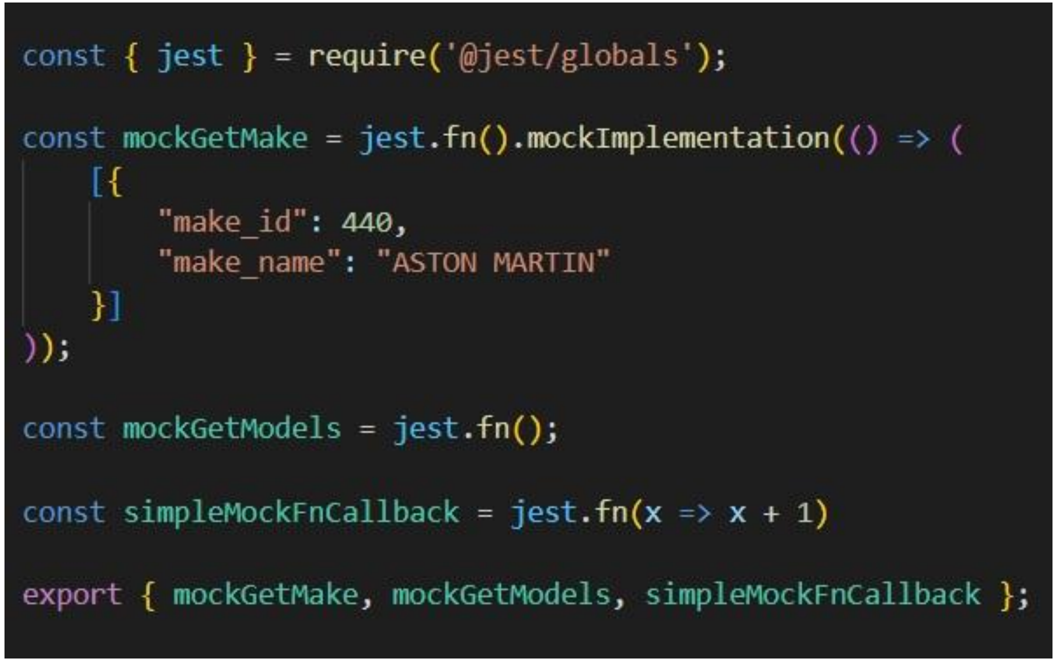
Jest Components: Mocks pt. 1

- Jest Mock Functions, also referred to as spies, allow you to spy on the behavior of your mocked function, seeing how often it has been called, and with what values. Mocks are not global.
- You will need to import them into test files like so:
  - **const { jest } = require(‘@jest/globals’)**
- To create mock functions in jest using `jest.fn()`, you do not pass a callback function into mock; it will simply return `undefined` and will wrap and execute that callback when called.

Jest components: Mocks pt. 2

Let's check out three examples of various ways to use jest mock functions.

- **mockGetMake**
- **mockGetModels**
- **simpleMockFnCallback**



```
const { jest } = require('@jest/globals');

const mockGetMake = jest.fn().mockImplementation(() => (
    [{
        "make_id": 440,
        "make_name": "ASTON MARTIN"
    }]
));

const mockGetModels = jest.fn();

const simpleMockFnCallback = jest.fn(x => x + 1)

export { mockGetMake, mockGetModels, simpleMockFnCallback };
```

Guided Walk-Through:

Writing Unit Tests with Jest

Let's head over to GitHub for some additional examples on creating and running unit tests with jest.

Break Time

Modern Engineering

API Testing with SuperTest

Testing HTTP Requests without HTTP Requests

So far, we've tested our express APIs manually using a series of Postman queries. This could be considered manual, end-to-end testing, the most basic and common level of testing something as you go.

Now, we'll use the JS library **supertest** to simulate these requests and parse through the responses in code!

Guided Walk-Through:

Setting Up Express to be SuperTested

Using supertest requires a bit of set up on the application side to create an easily exportable version of the server.

Let's set up the most basic possible server with supertest attached by following along in this git repository.

Group Exercise:

Supertest defeats Postman!

90 minutes

Now that we've added supertest to our toolbox, let's replicate the manual E2E testing we've done in Postman by creating a suite of Supertest tests on the CRUD routes for our [ToDo Express API Application].

Solo Exercise:

SuperTest Your Application

90 minutes

Now that we've seen SuperTest applied to the To-Do Express API, set up your personal project Express API with a suite of tests around the basic CRUD routes using the same pattern.

Solo Exercise:

Mid-Course Feedback Survey

10 minutes

- Please take a moment to give us your feedback about your GA Modern Engineering Fundamentals mid-course experience!
- Follow the link: https://bit.ly/41Ky9Z2